# Scalable Translation Validation of Unverified Legacy OS Code

Amer Tahat*, Sarang Joshi†, Pronnoy Goswami‡, Binoy Ravindran§
Virginia Tech University, Blacksburg, VA
Email: { antahat *, sarang87†, pronnoygoswami‡, binoy§}@vt.edu

***Abstract*—Formally verifying functional and security properties of a large-scale production operating system is highly desirable. However, it is challenging as such OSes are often written in multiple source languages that have no formal semantics – a prerequisite for formal reasoning. To avoid expensive formalization of the semantics of multiple high-level source languages, we present a lightweight and rigorous verification toolchain that verifies OS code at the binary level, targeting ARM machines. To reason about ARM instructions, we first translate the ARM Specification Language that describes the semantics of the ARMv8 ISA into the PVS7 theorem prover and verify the translation. We leverage the radare2 reverse engineering tool to decode ARM binaries into PVS7 and verify the translation. Our translation verification methodology is a lightweight formal validation technique that generates large-scale instruction emulation test lemmas whose proof obligations are automatically discharged. To demonstrate our verification methodology, we apply the technique on two OSes: Google's Zircon and a subset of Linux. We extract a set of 370 functions from these OSes, translate them into PVS7, and verify the correctness of the translation by automatically discharging hundreds of thousands of proof obligations and tests. This took 27.5 person-months to develop.**

***Index Terms*—Formal Verification, Linux OS, Google Zircon**

## I. Introduction

Operating systems are among the most critical parts of a computer system. Faults in their kernels, for example, may leave the entire machine vulnerable to serious functional and security failures. Thus, they require high levels of reliability. Formal verification techniques can help achieve high levels of reliability. However, formally verifying that an OS behaves as expected is challenging [1]–[3]. For instance, many production operating systems are large (e.g., in the hundred thousand LOC range or even bigger). This requires solving many computationally challenging problems for formal verification tools, such as extracting the control flow and function call graphs from code, capturing code semantics, and validating those semantics against their original intended meanings, among others. Prior work demonstrates that this often requires many person-years of work [3], [4]. For example, the seL4 OS verification effort, which involved 10K source LOC, took 20 person-years [5]. Moreover, OS source code is commonly written in languages (e.g., C) that lack formal semantics – a prerequisite for formal reasoning over code properties at the source level. Furthermore, many OSes are written in a combination of a high-level language such as C and low-level assembly code, and often include library code, which may be written in (yet) another language. Thus, verification efforts must formalize the semantics of all languages used – a daunting task.

Operating systems, similar to many large-scale software systems, quickly become legacy. Enterprise-class organizations and open-source communities that develop OSes often invest significant resources. Due to this investment, such code bases are rarely discontinued; rather, they are continuously enriched with new functionality, patched to add new security features, and ported to new hardware and maintained, all over long life cycles. Any formal verification technique that targets such software must inevitably deal with code maintenance: how to cost-effectively verify when changes are made?

These difficulties have, by and large, caused many large-scale production OSes (e.g., Linux, Android) to be out of scope of formal verification techniques. In this paper, we consider formally verifying OS code at the binary level. Binary verification sidesteps some of the difficulties of source code-level verification, such as the necessity for formal semantics of multiple source languages. However, binary verification comes with its own set of challenges, such as lack of type information, data structures, and control-flow information. These difficulties make it hard to reason about code properties at the binary level. Translating the binary into a higher level of abstraction that is easy to reason about (e.g., high-order logic in a theorem prover)would circumvent such difficulties. However, this requires verifying the integrity of the translation.

Formal verification using theorem proving techniques is known to provide the highest levels of assurance [1], [6], [7]. It can establish a program's correctness using abstract theorems that are independent of the size of the state space, increasing effectiveness for verifying large code bases [1], [7]. However, state-of-the-art in theorem proving-based verification has by and large focused on verifying at the source code level. For example, the seL4 effort [1], [2], which uses theorem proving, assumes that complete high-level source code of the OS is available to the verifier in a subset of the C language, called $C_0$. [4] presents a verification toolchain that targets a typed assembly language, which is transformed into a typed machine language to generate a safe binary. [3] presents the formal proof for a compiler called CompCert, but restricts it to a subset of C called C-light [8]. [9] presents an approach for designing a new OS kernel from scratch that is verifiable using SMT solvers, but the approach scopes out verifying legacy operating systems. [10] establishes that seL4's binary code is

equivalent to its $C_0$ source, but is restricted to the already verified seL4's $C_0$ code.

It is compelling to develop a theorem prover-based verification toolchain that can formally verify OS binary code. Such an approach must overcome significant challenges. First, since the toolchain's input is binary code, reasoning about properties of the code will depend on the target machine architecture. Therefore, building the toolchain must start with a valid formal model of the target machine's instruction set architecture (ISA) in a theorem prover. This problem is significant by itself [11]–[15]. However, it is a critical step toward building a trustworthy decoder that can decode binary streams into their machine instruction representations, enabling verification of program binaries (for that machine) in a subsequent phase. Second, the formalization of the targeted ISA's should be elegant, readable, and as close as possible to their machine-vendor-specific representations [11], [13], [14]. This can reduce verification costs. To address these challenges, we present a verification toolchain that utilizes the analytic power of state-of-the-art reverse engineering tools such as radare2 [16] and the rigor of a powerful interactive theorem prover, PVS7 [17]. We focus on a subset of the ARMv8 A64 ISA as the target architecture. The rationale behind this choice is that ARM recently introduced the ARM Specification Language (ASL) in machine-readable form [18], [19]. ASL supports dependent types that are also supported in PVS7. Moreover, ARM reports that ASL has been extensively tested (e.g., millions, in some cases billions, of test suites) [19].

Our verification toolchain and workflow are illustrated in Figure 1. To reason about ARMv8 instructions, we first translate ASL into PVS7 (Section II). We introduce a new logical framework that captures the precise operational semantics of ARMv8 instructions in PVS7 with a small trust base. The framework uses PVS7's parameterized generic theory declarations and dependent types, which enables an almost one-to-one translation between ASL and PVS7.

To verify the ASL to PVS7 translation, we developed a lightweight formal validation technique, folded into a tool that generates large-scale instruction emulation, test lemmas and their proof scripts in PVS7, whose proof obligations are automatically discharged (Section III). We built this tool on top of the Unicorn CPU emulation framework [20] and our ASL to PVS7 formal model. The approach incorporates generic theories to reduce the cost of proofs of the test lemmas, resulting in savings of thousands of proof LOC. We leverage the radare2 reverse engineering tool to decode ARM binaries and lift them into PVS7 models (Section IV). We reuse the formal semantics of ARMv8 instructions, combined with a formal translation of program control flow and function call graphs to build up the formal semantics of code. We develop an efficient validation technique for the radare2 to PVS7 translation (Section V). The approach relies on using ASL to extract what we call reverse decoding dictionaries. We also develop a methodology to validate control flow translation integrity from radare2 to PVS7. To demonstrate our verification methodology, we apply the technique on two OSes: Google's Zircon and a subset of Linux. We extracted 127 functions from the Zircon binary, translated them into PVS7, and validated the translation along with 243 functions from Linux. To the best of our knowledge, this is one of the first works that attempts to formalize system software containing such large amounts of source code.

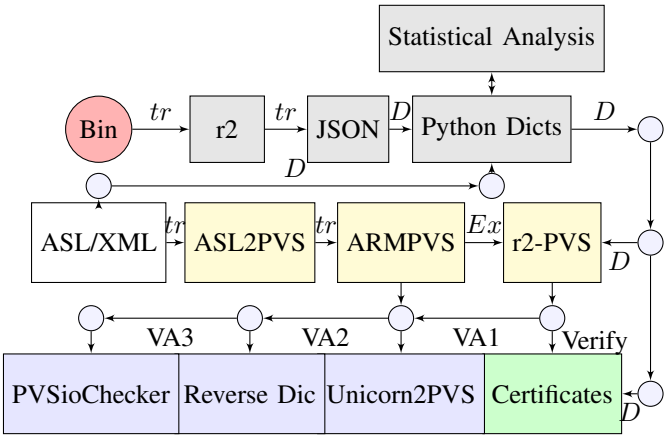



Fig. 1: Toolchain Workflow. $r2$= radare2, $tr$= Translation, $D$= Data transfer, $Ex$= Theories Export, VA= Validation.

## II. Formalizing ARMv8.3 A64 in PVS7

We present a new theoretical framework to capture the imperative nature of ASL in PVS7. We first summarize three features of PVS7 that we use to build our framework. Then we discuss how we use them to capture ASL semantics.

### *A. Dependent Types, Theory Parameters, and Generic Theories of PVS7*

Dependent type declarations and theory parameters of PVS7 simplify the formalization and reasoning at the binary level as it allows efficient representation of polymorphic bit vectors that are common in binary analysis. In PVS7 notation, a theory name is declared first; then the keyword `Theory` is followed by the theory parameter (if they exist). After that, the developer can declare other elements such as constants and variables. For instance, the bit vector type in the PVS7 library is declared in a theory called `bv`. This theory has the parameter `n`, which is a natural number that represents the length of the vector. In particular, `bv: Theory [n : Nat]` is a parametric theory in which we can see the type declaration `bvec[n] : Type = [ below(n) -> bit]`. Here, the type `bvec[n]` is said to be a dependent type on the theory parameter `n`. In this theory, it is defined as a function from the type of all natural numbers that are less than `n` to the type `bit`, which is either 0 or 1. As a result, one can declare two constants of multiple bit vector instances of different lengths only by instantiating the parameter `n` appropriately. A theory may use more than one parameter. Nonetheless, they have to be instantiated fully each time a parametric theory is used. Partial instantiations are not allowed. To solve this problem, PVS7 provides an enhanced mechanism for generic theory declarations. A generic theory in PVS7 allows users

to build formal modules in an objected-oriented style and use them efficiently as many times as they desire. Specifically, a new theory "Object" can be generated from a generic one using theory declarations. For instance, to declare two new theories, say `B` and `C`, from a previously declared theory, say `A`, that has two unspecified constants, say `a1, a2` of type bit vector of length $n$, one can efficiently use two lines of PVS7 code: `B : Theory = A with {{ a1 := bv[2](0b01) }}` and `C : Theory = A with {{ a1 := bv[3](0b101), b:= bv[2](0b10)}}` to declare `B` and `C`. Here, B is declared as a theory that is identical to A except `a1` is a constant of the concrete value `bv[2](0b01)`. Observe that `B` partially instantiates `A` as `a2` is left unspecified. `C`, however, fully instantiates `A`, as both `a1` and `a2` are instantiated with concrete values. Also, note that `a1` has been populated with two bit vector values of different lengths in the declaration of `B` and `C`.

### B. Imperative Aspects of ASL in PVS7

Our basic idea is to build a generic parametric theory for each instruction of ASL in PVS7. Then we load them with concrete values to produce different instances of bytecode. To explain our approach, we first briefly discuss the ASL code structure. ASL is a machine-readable specification language developed by ARM [21] that replaces 6K pages of ARM manual [22]. It is imperative and supports dependent types, exceptions, conditionals, as well as loops. We restrict our formalization to a subset of the ARMv8.3-A64 architecture that we identified to be used frequently in our verification targets (Zircon and Linux). The subset belongs to the following instruction classes: `Data Processing (Immediate)`, `Data Processing (Register)`, `Loads`, `Stores`, and `Branches`. These instruction classes are specified in XML files. The files also use a shared library of common functions that are declared independently in the `Shared Pseudocode Functions` XML file. Each instruction

```
|31 |30:29| 28:24| 23:22|21|20:16|15:11|10:5|4:0|
|sf |0  0 | 01010| shift| 0|Rm   | imm6| Rn |Rd |
      opc                  N
```

Listing 1: ASL-XML ands-log-shift bits diagram

XML file consists of three major parts: i) a diagram that illustrates the bit format of each instruction, (such as Listing 1); ii) a decoding part in which variables are declared, initialized, or assigned values that are extracted from the diagrams and iii) an operational part which computes the post-state.

### C. Mapping from ASL to PVS7

Each instruction is expected to operate on some information extracted from the current machine state; all instructions' generic theories have an `arm-state` as a theory parameter. This parameter captures the machine's pre-state `p` before executing the current instruction. The class of this parameter is called `arm-state`. We implement `arm-state` as a generic theory. This theory has unspecified types and constant declarations. We declare `s` as the machine state type which is a record type with fields representing all 31 general purpose registers `X(n)`, the stack pointer `SP`, `PSTATE registers`, the program counter `PC`, and a memory function `Mem`. According to our targets, `Mem` is defined from natural numbers to bvec[64] with fixed normal access type and 8 as access size. The first challenging part of the translation of ASL into PVS7 is to correctly obtain the bit format of each instruction

```
diag: Type+ =
[# sf:bvec[1],opc: bvec[2],Fixed1:bvec[5],shift:bvec
    [2],N:bvec[1],Rm: bvec[5],  imm6:bvec[6], Rn:
    bvec[5], Rd:bvec[5] #]
```

Listing 2: PVS and-log-shift bits diagram

(as shown in Listing 1) in the theorem prover. This is error-prone and impractical if done manually. Thus, we directly extract the diagrams from the XML files and translate them into a PVS file. In XML, the diagrams include field names, indices, and possibly some fixed values. We translate them into a type called `diag` of PVS's record type. PVS allows partial updates for record types. For example, if `x1` is a record that has two fields of natural numbers `a` and `b`, then the expression `x2 = x1 with [a:= 5]` means that `x2` is identical to `x1`, except that the value at field `a` is now 5. Fields of the `diag` type are all dependent types on their indices, for which our ASL to PVS translator automatically calculates the length of each field. Listing 2 illustrates this. In each generic theory, we define another unspecified constant called `Diag` of type `bvec[32]`. We load `Diag` with the bit vector of interest from the binary code using generic theory declaration. We also define the symbolic version of the loaded value of `Diag`, which we call `v` so that we can use it in the decoding part of the generic theory. `v`'s type is dependent on two slicing functions, `bt` and `bts`. For example, in Listing 3, the expected value of the field `Rd` is the slice 0-4 of the vector `Diag`, which is identical with the ASL expectation

```
v:diag=(# sf:=bt(Diag,31),opc:= bts(Diag,29,30),
Fixed1:= bts(Diag,24,28), shift:= bts(Diag,22,23),
N:= bt(Diag,21)          , Rm:= bts(Diag,16,20),
imm6:= bts(Diag,10,15)   , Rn:= bts(Diag,5,9),
Rd:= bts(Diag,0,4) #)
```

Listing 3: PVS ands-log-shift diagram load

(Listing 1). Moreover, we use `v` in the decoding and the operational parts of the ASL code. For instance, the ASL declaration `integer d = UInt(Rd);` can easily be declared in PVS as `d:int= UInt(5,v.Rd)` where we implement `UInt` as an (unsigned integer) function in PVS.

### D. Operational Sequential Semantics

For the operational part of the ASL translation, we define a new dependent type called `ASL(p)`. This type conveniently helps us to capture the state of the ASL mutable variables, `arm-state` changes, side effects, and the machine post state after executing the instruction. Our idea is to define each statement of ASL code as dependent on the previous one and the theory parameter `p` (the machine pre-state). This allows us to produce an almost one-to-one translation for each line of ASL code in PVS7 (see

Listings 4, 5). We use the standard `throw` function of PVS7 to throw exceptions when exceptions are raised in ASL. For example, the ASL line of code `if sf ==0& imm6<5> == 1 then ReservedValue();` raises the exception `undefined-fault`. Thus, we translate it to `if v.sf = bv[1](0b0)&(v.imm6(5) = True) then throw (''ReservedValue().undefinedfault'')else sts1 endif.` ASL also has loops. In our model, functions that contain loops in ASL are either represented

```
sts3: ASL(p)= sts2 with [operand1:= p.X(n)]
sts4: ASL(p)= sts3 with [operand2:=ShiftReg(64,p.X(
   m), shift_type, shift_amount)]
sts5: ASL(p)= if invert then sts4 with
                  [operand2:= NOT(sts4.operand2)]
               else sts4  endif
sts6: ASL(p) = Cond sts4.op = LogicalOp_AND -> sts5
    with [result := AND (sts5.operand1, sts5`
   operand2)],
                    sts4.op = LogicalOp_ORR ->
   sts5 with [result := OR  (sts5.operand1, sts5.
   operand2)],
                    sts4.op = LogicalOp_EOR ->
   sts5 with [result := XOR (sts5.operand1, sts5.
   operand2)] EndCond
% post state
p1:s = if sts6.setflags then p with [.PSTATE.NZCV:=
  let result_63 = field(64, sts6.result, 63 ,63)
  in bv[2](0b00) o IsZeroBit(64,sts6.result) o
    result_63]
       else p  endif
post: s = p1 with [.X(d) := sts6.result]
```

Listing 4: PVS ands-log-shift Operational

by classical tail recursion or by means of the standard `for` function in PVS lib. Instructions that belong to the same class may share a lot of the common operational code.

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m,
                  shift_type, shift_amount);
if invert then
            operand2 = NOT(operand2);
case op of when LogicalOp_AND
                  result = operand1 AND operand2;
          when LogicalOp_ORR
                  result = operand1 OR operand2;
          when LogicalOp_EOR
                  result = operand1 EOR operand2;
%Post state
if setflags then PSTATE.<N,Z,C,V> =
       result<datasize-1>:IsZeroBit(result):'00';
X[d] = result;
```

Listing 5: ASL ands-log-shift Operational

The actual differences in these cases are in their diagrams. We exploit this property in our translation to reduce the trust base. In particular, we practically generate the diagrams and most of the decode parts fully automatically from XML, while we use a semi-automatic approach for the operational parts. For instance, in the `log-shift` class of instructions, it was sufficient to develop one of the instruction's operational code manually (≈20 ASL LOC). The other seven instructions of this class share identical operational code. Thus, the translator will automatically copy the shared part to the rest of the members and generate their generic theories, thereby potentially minimizing the errors that may occur in a manual approach. However, for some instructions such as `Load` and `Store`, this approach might complicate the proofs as, in some cases, large parts of the shared code might be unreachable due to case analysis. Thus, for these cases we manually optimize the formalization for each instruction to enhance the performance of the proofs.

## III. Validation of ASL to PVS Translation

### A. Test Case Generation

Our basic idea is to use large-scale litmus testing for each formal instruction in our model. We use a CPU emulator called Unicorn to generate large test suites. In Unicorn, one can manually set the instruction bytecode and the pre-state values for the registers as well as the boundaries of the memory regions that will be used in the emulation. Unicorn reports the post-state of the emulation as output. We automate this process to generate a large number of test suites and map the pre- and post-states into PVS7 lemmas. Then we generate the proofs of the lemmas highly automatically. We refer to our tool as UniVS7. Naïve generation of a random string of length 32 will not give us valid bytecode for emulation. To address this problem, we extract the format of the bytecode from ASL's XML diagrams. We then use a constrained but random generation of the slices that are parametric in the diagrams. For instance, in Listing 1, we use standard bounded random generation function of integer values. The function follows a uniform distribution; we use it to generate the non-fixed fields of XML diagrams such as `d`, `m`, `n`, and `imm6`, and the values of `Rd`, `Rn`, and `Rm`. The integral values must not exceed previously set conditions. For example, `d` must not exceed 32, so we generate the values randomly in this restricted domain. All integral values are then converted to binary strings. We then concatenate the whole strings into a single bit string. The resulting string is converted into Unicorn bytecodes, which can then be emulated. For other instructions such as load/store, test case generation is more complex and requires additional mathematical insights. For instance, we have to develop and solve an algebraic inequality over the random variable such that we guarantee that the generated values will belong to the designated memory region. Otherwise, Unicorn will issue an error message, indicating that the pre-defined memory boundaries have been violated and no post-state can be generated. For example, assume that we devote `[0x10000 - 0x200000]` to be the memory region for emulation. Moreover, assume that we have to generate a value, say $x$, for which an instruction will apply some binary arithmetic to obtain a new address, such as `Rn + Sign-Extension(x)`. Before it runs the emulation, Unicorn must check that this new address belongs to the emulation region. Hence, we have to solve the inequality `0x10000 <= Rn + Sign-Extension(x) <= 0x200000` for $x$. Typically, the algebraic solution is a bounded interval on $x$, which will represent the boundaries of the random generation for $x$. In all generated test suites using this technique, we detected no failures during Unicorn's emulation.

```
1 ANDS_LOG_SHIFT_8A0A1C93: THEORY BEGIN
2       IMPORTING rsl@log_shift
3 p:  s = init with [ .X:= unicorn_pre_state ]
4
5 new_test_obj: Theory = log_shift[p]{{
6           Diag:= 0b11001001001110000101000001010001,
7           addr:= 0x10000 }}
8
9 test1: lemma let X_post= p.X with [.X:=
      unicorn_post_state ] in
10              let p2=  p with [.X:= X_post] in
11                new_test_obj.post = p2.post
12
13 %UniVS7's auto generated Proof-Lite scripts:
14 %|- X_sts_TCC*       : PROOF
15 %|- test1_TCC1       : PROOF (eval-formula)  QED
16 %|- test1 :PROOF (log-shift) QED
17 END ANDS_LOG_SHIFT_8A0A1C93
```

Listing 6: ands_log_shift UniVS7 generic test format

*1) Type Check Obligations:* After test suites have been generated, we automatically transform them into PVS7 in the form of *test lemmas*. Each test is written in a new theory that imports the generic theory of the instruction subject of validation. We define a new object of the instruction with a new value for the theory parameter `p`, which represents the pre-state, and a new value for the `Diag` element of the generic theory. UniVS7 will automatically populate `p` and `Diag` with these values from the intermediate files. We write a test lemma for each test theory that states that the post-state of the instruction that was generated by our translator matches the expected value of the post-state generated by Unicorn. These lemmas must then be proved (by UniVS7), which we discuss in the next subsection.

### B. Proof Automation

We design our formalization so that instructions that belong to the same class share most of the same proof scripts. For instance, the script of Listing 6 (lines 13-16) can automatically discharge tens of thousands of proof obligations generated for 2500 test theories for the entire ASL ARMv8-A64 log-shift class of instructions. To illustrate our proof mechanism that achieves such a degree of automation, we discuss two types of proof obligations that are generated by our tool UniVS7. We exploit PVS7's powerful dependent type mechanism to generate and discharge the necessary proof obligations highly automatically. For instance, in PVS7, if we have a concrete instantiation of any dependent type to a constant, the PVS type checker will attempt to generate and prove a proof obligation that assures that the new concrete value passes the dependent type membership conditions, called type check correctness conditions (or TCCs). For example, suppose we declare `x` in PVS7 to be a constant of type `bvec[2]`, which was later populated with the value `0b011`. The type checker will help us to discover this bug as it will generate an unprovable TCC to show that `UInt(x) < 4`. However, if we had populated `x` with a valid member of the type `bvec[2]`, then the corresponding TCCs will automatically be discharged. UniVS7 will invoke the theorem prover to assure that the loaded values in the model from the test generation phase pass the type conditions of the record types that were obtained from the ASL to PVS translation phase (see for example, Listing 2). The type checker will automatically generate thousands of proof obligations. Moreover, UniVS7 will install our proof scripts, for example, Listing 6 (lines 13-16), to discharge them for the entire test suit atomically.

*1) Operational Lemma Proof Obligations:* UniVS7 generates a second type of obligations that are also necessary for proving the test lemmas, for example, Listing 6 (line 9). The greater the number of lemmas UniVS7 proves, the greater is the trust that we gain in the operational semantic translation of ASL to PVS7. However, proving a large number of test lemmas that mimic the behavior of an ARMv8 instruction in any theorem prover is challenging. For example, an ARMv8 instruction may call, on average, one hundred functions that may consist of thousands of LOC [23]. However, only part of the PVS specification language is executable in its evaluator [24]. Therefore, using only a one-step expansion mechanism of a theorem prover may not be feasible [13]. To address this challenge, we purposefully formalize most of the required support functions of ASL using PVS7's executable specifications. For example, for `and-log-shift` instruction, the functions `ShiftReg`, `IsZeroBit`, `UInt`, `AND`, `OR`, and `XOR` are all implemented as executable functions. Moreover, as illustrated in Section II-D, we label each statement of the code with a symbol such as `sts-i`, `p`, `p1`, and `post`. Thus, to prove a test lemma according to our model, we only need to apply expansions to the statement symbols and to the vector `v` that receives the loaded binary from the constant `Diag` in each test declaration, as explained in Section II-C (Listing 3), as well as the field names for any instruction (for example, see the `log-shift` proof strategy in Listing 7; lines 1-3

```
1 ( defstep log-shift () (then
2   (expand-par "p" "post" "p1" "sts6" "sts5" "sts4" "
      sts3" "sts2" "sts1" "p" "X_sts" "init" "d" "v" "
      n" "invert" "m" "shift_type" "shift_amount")
3   (eval-formula 1)) "" "")
4
```

Listing 7: log-shift class proof strategy

the user defined strategy `(expand-par)` expands its parameters sequentially). These expansions will load the concrete values in place of the symbols. The prover will then reduce the proof problem into fast numerical functional evaluations of the support functions that are called in the formalization of the instruction (see Listing 7; line 3). These numerical evaluations are too fast (millisecond scale) when compared to lazy expansions. Using an executable evaluation of a formula in a proof of a PVS lemma is called *numerical reflection* [25]. It is usually invoked using a proof strategy (`eval-formula` in line 3 of Listing 7). Since members of the same class often share these symbol names, UniVS7 can reuse the same Proof-Lite script [7], [24], such as Listing 6 (lines 13-16), for all the class members. This allows discharging tens of thousands of test lemmas with only one proof command for each test suite.

## IV. RADARE2 TO PVS7 TRANSLATION

### A. Analysis of Zircon & Linux Instruction Classes

There are two questions that we have to answer before we start the formalization of large legacy OS binary code such as Zircon and Linux. First, which set of OS functions should we start with? Second, to be most productive, how can we increase reusability of our formalization of the instructions (i.e., using the formalized instructions of one OS code in reasoning about another OS code)? For the first question, we decided to start with terminal functions. These functions do not call other functions. They are invasive in our targets; all non-terminal functions call some of them by definition. Thus, they represent the starting point of the formalization of functions in the target and include system calls. We also added another filter on our verification space: we decided to first extract terminal functions that don't have cycles in their control flow graphs. If our toolchain does not scale on such functions, it is unlikely to scale on more complex ones, and we may consider a change in our design. In our two targets combined, the number of such functions is 370, with a total size of ≈1000 instructions. This is ≈20% of the terminal functions in the two targets. For a perspective, this is just over a quarter of the size of the functions formalized in [10], which required several person-years to develop. To answer the second question, we conducted a statistical analysis on a disassembled version of the two OS binary codes (≈600K instructions combined). We analyzed the number of functions of each OS that use different classes of instructions. Figure 2,

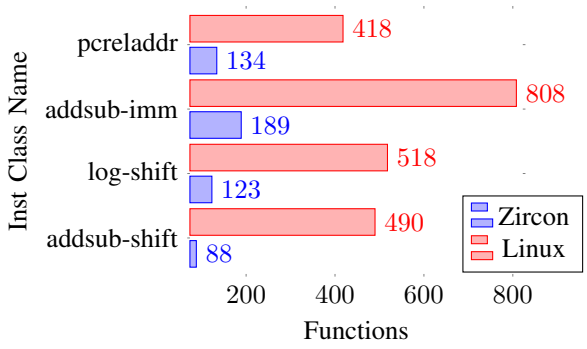



Fig. 2: Instructions classes usage in Zircon and Linux.

which shows part of the results[1], reveals that the `pcreladdr` instruction class appears in the formalization of 134 functions of Zircon's terminal functions, whereas it is used by 418 terminal functions in Linux. Our complete analysis revealed that Zircon target uses 30 instruction classes and Linux target uses 33 classes. We gave higher priority to formalizing the instruction class with higher usage count across both targets. We also analyzed the OS codes in the reverse direction: i.e., the set of instruction classes that are used by a given set of OS functions. Our analysis revealed that a list of 10 classes covered 100% of the 370 functions that we extracted from the two OSes. Thus, it was sufficient to restrict our ARMv8-A64 model (Section II-D) to these instruction classes.

[1]Our artifacts are available at https://github.com/ssrg-vt/renee-artifacts

### B. Function Formalization in PVS7

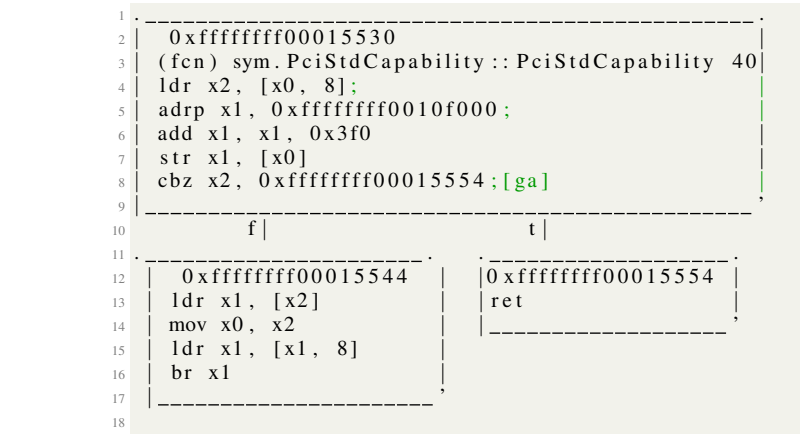
*1) Auto-Generation of ARM Decoder into PVS7 Logic:* Given the large number of bytecodes in our targets, we require a reliable decoder that can decode a bytecode into its instruction theory without errors. Manually writing a decoder is highly error-prone. ASL contains all the bit patterns of ARMv8-A64 instructions. They are organized in machine-readable tables that link a pattern into its corresponding instruction's XML file name. Thus, we extract these tables from ASL and convert them into Python dictionaries. We then develop a parser that matches a bit string with the bit pattern from our dictionaries, thereby obtaining the correct decoding of a bitstream. The parser produces a PVS7 instruction name that is designed to be identical to the corresponding instruction XML file and the PVS generic theory names that we generated in the ASL to PVS translation (Section II).

```
1  GET_PIPE_INFO_FFFFFF80081119F8[(IMPORTING arm_state)
       p:s]:THEORY BEGIN
2
3  movz.0:Theory= movz[p]
4          {{Diag:=b11010010100000000000000000000000,
             addr:=0xffffff80081119f8 }}
5
6  ret.1:Theory= ret[movz_0.post]
7          {{Diag:=b11010110010111110000001111000000,
             addr:=0xffffff80081119fc }}
8
9  B_post: s = ret_1.post
10
11 %|- *_TCC*: PROOF (eval-formula) QED
12 END GET_PIPE_INFO_FFFFFF80081119F8
```

Listing 8: Generic theories based basic block translation

The advantage of this technique is that we have a small trust base. For instance, ARM has intensively tested ASL. Thus, ASL is likely the most reliable specification of a mainstream ISA ever published [21]. Our code to extract the decoding dictionaries and match the bytecode pattern with the instruction is only 350 (Python) LOC, which we add to the trust base.

*2) Basic Block Translation:* Our radare2 to PVS7 translator extracts the bytecodes and the addresses from radare2's JSON basic block representation of a given function (for example, see Listing 9). Subsequently, the translator will translate each basic block into a PVS7 theory with an `arm-state` parameter (Listing 8). This parameter will capture the pre-state of the machine before executing the block. Moreover, the translator will generate a new PVS7 valid instance of the object that represents that particular bytecode. The translator does this by instantiating the generic theory's `Diag` and `addr` attributes from the JSON files. Furthermore, for continuity, we pass the current instruction's post state as the theory parameter of the following instruction's theory declaration. Finally, this new block theory file will have its `B_post` element which captures the machine post-state after executing the function. For example, Listing 8 shows a basic block extracted from Linux's `GET_PIPE_INFO` function into PVS7. In lines 3 and




Listing 9: PciStdCapability::PciStdCapability radare2 CFG

6, we can see the familiar names `movz` and `ret`. The translator adds the class name and an index to the end of each instruction name. This disambiguates these names in the theorem prover when they are used in the same PVS file. Line 1 of Listing 8 shows the `arm-state` theory parameter `p`. Finally, line 9 presents the block's post state.

*3) Formal Translation of CFG:* We adopt radare2's control flow graph (CFG) representation in our model. In this representation, we can see a function as a graph for which the basic blocks are the nodes and the jumps between them are the edges (for example,see Listing 9). After translating each basic block into a PVS7 file, our toolchain will generate a PVS main theory file for each function (Listing 10). The file includes an import statement to each basic block theory. In addition, the toolchain will extract each path on the CFG and translate it into a possible post-state of the main PVS file. For example, the Zircon function `PciStdCapability::PciStdCapability` has three basic blocks and two possible paths, represented in Listing 10 lines 7, and 12. The first path indexed by `_post0` is interpreted as the leftmost path of Listing 9. However, it reads in a backward direction (i.e., the innermost state is the first and the outermost is the last). In particular, lines 8-10 indicate that, after executing the basic block at `FFFFFFF00015530`, the flow will pass the block's post state `B_post`

```
1  PCISTDCAPABILITY_PCISTDCAPABILITY_MAIN [ (IMPORTING
       arm_state ) p : s ] : THEORY
2  BEGIN  IMPORTING
3 PCISTDCAPABILITY_PCISTDCAPABILITYFFFFFFFF00015530,
4 PCISTDCAPABILITY_PCISTDCAPABILITYFFFFFFFF00015544,
5 PCISTDCAPABILITY_PCISTDCAPABILITYFFFFFFFF00015554
6
7 PCISTDCAPABILITY_PCISTDCAPABILITY_post0 : s =
8 PCISTDCAPABILITY_PCISTDCAPABILITYFFFFFFFF00015544[
9 PCISTDCAPABILITY_PCISTDCAPABILITYFFFFFFFF00015530[
10 p ]. B_post ]. B_post
11 %
12 PCISTDCAPABILITY_PCISTDCAPABILITY_post1 : s =
13 PCISTDCAPABILITY_PCISTDCAPABILITYFFFFFFFF00015554[
14 PCISTDCAPABILITY_PCISTDCAPABILITYFFFFFFFF00015530[
15 p ]. B_post ]. B_post
16
17 END PCISTDCAPABILITY_PCISTDCAPABILITY_MAIN
```

Listing 10: PciStdCapability::PciStdCapability PVS7 CFG

to the block at address `FFFFFFF00015544`.

## V. RADARE2 TO PVS7 TRANSLATION INTEGRITY

Unlike single instruction validation, using a large test suite of concrete emulations for the validation of a large piece of code is computationally challenging due to state space explosion. Therefore, for large code sizes, we develop a practical yet rigorous validation technique called *reverse dictionary*, which we use in two ways.

First, we use ASL in the reverse direction of the decoder we call this technique `Reverse Decoding Dictionary`. This tool parses PVS7 basic block models of each function and extracts the names of the instructions, bytecodes, and the addresses from the `Diag` and `addr` attributes, and stores them in a dictionary. According to the decoder's instruction naming convention, the tool then retrieves the name of the ASL instruction's XML file. The tool then extracts the bit format diagram from it. The XML bit format has fixed and variable fields. Therefore, the tool does two assurance checks. First, it matches the `Diag` bit vector with the XML bit format. The bit vector must necessarily satisfy the pattern if the translation is correct. This check ensures that the `Diag` instruction selections have not changed while transferring the data from radare2 to PVS7. Second, it reconstructs the bytecode by filling the variable parts of the XML diagrams with the corresponding slices from the extracted bit vectors of `Diag`, leaving the fixed parts as is. The constructed bytecode is then compared with the bytecode extracted from JSON. This check proves that the entire bytecode has not changed while transforming the data from radare2 to PVS7. Observe that extracting the `Diag` bit vector from PVS7 and only matching it directly with radare2's bytecode does not ensure that our decoder (Section IV-B1) has selected the right instruction PVS7 file names. Thus, both steps are necessary to support the integrity of the translation. For more efficiency, our tool checks these tests first in Python. For additional formal assurance, the theorem prover, as illustrated in Section III-A1, will produce and prove all the TCCs for each instruction's theory declaration.

Second, we also verify that the program control flow graph did not change during the translation. We develop a technique called *CFG Reverse Dictionaries*, wherein we parse PVS7 main files, as in Listing 10, and extract the control flow paths. Since each path has a counterpart in the radare2 to PVS7 translation tool, we compare the two paths and detect any mismatch. In our case studies, we did not detect any CFG mismatches, increasing our trust in the translated CFGs' integrity. We also provide a test lemma for each branching instruction for each basic block in PVS. We formalize a lemma similar to those of Section III-B1. However, we do extract the next addresses from radare2 and then prove in PVS7 that the post state program counter register will be updated to the expected destination address. Also our validation tool assures that the addresses in radare2 are identical to their PVS7 counterparts.

## VI. Verification Statistics

In the targets, there was a total of five languages used at higher levels such as C, C++, assembly, and some Python and Shell scripts, totaling 171K high-level source LOC to analyze. Translating such large code to a formal language is out of the scope of any known formal method techniques. Our approach, in contrast, enables the possibility of extracting functions of interest from a target code and then automatically formalizing it based on its structure. For example, we extracted all functions (from both targets) that have a tree structure for its CFG and terminal functional call graph. Thus, the conversion step into a JSON IR representation has resulted in 25K JSON LOC, which is practically feasible to parse.

On a machine with Intel Core i7-4910MQ processor running at 2.9 GHz with 15.6 GiB memory, our toolchain parses these LOCs efficiently and produces a type-checked PVS7 model in less than two hours ($\approx$ 9300 LOC PVS7 and Proof-Lite scripts). For instance, the Linux sample was translated in $\approx 16$ minutes and it was type-checked in $\approx 48$ minutes ( $\approx 6950$ PVS7 and Proof-Lite scripts ), where as Zircon's translation time was $\approx 10$ minutes and its type-check time was $\approx 35$ minutes ( $\approx 2253$ PVS7 and Proof-Lite scripts ). In addition to ASL, we also trust the reverse engineering tool and the CPU emulator. Given our time and resource constraints, for appropriate scoping of our work, we chose to put these tools in our trust base. The remainder of our trust base includes $\approx 1430$ LOC of Python and shell scripts. These scripts automate the entire analysis that we do in our toolchain, such as parsing $\approx 0.6$ million LOC, searching for the targeted functions, and translating them into PVS7.

UniVS7 tests lemmas and the discharged TCCs are $\approx 55K$, distributed into 30 test suites, in average each test suite ran in $\approx 20$ minutes; each test suite contains $\approx 1800$ test cases. We use these tests to validate our semi-automatic ASL to PVS7 translation (Section III).

## VII. Past and Related Work

Translation verification has been extensively studied in the past. This literature can be broadly classified into two categories: *verified compilation* [26] and *translation validation* [27], [28].

In the first approach, formal semantics for the source and destination languages are constructed at a high-level of abstraction such as HOL, and often specified inside a theorem-prover. A *refinement* relation is then established between specifications written in the two languages, using prover-generated proofs. A refinement between the specifications implies that every behavior of one specification is allowed by the other, essentially implying that the specifications are (behaviorally) equivalent. (Establishing a direct refinement between the two specifications is often difficult; in such cases, intermediate languages are usually introduced). Examples of this approach include [26], [29].

In the second approach, the output of each translation is shown to be equivalent to the corresponding input, in a post-translation validation phase. This equivalency is often established through a *simulation* relationship [30]–[32]. The fundamental difference between these efforts and ours is that we do not require the existence of the source code or its formal semantic model. In addition, the source codes of our targets are out of scope of these past efforts due to their large size and use of multiple languages.

Although originally, translation verification approaches have focused on source code, subsequent efforts have targeted binary code using precise formalized instruction set semantics (e.g., [23], [33], [34], [12], [13]). In general, these approaches are generic, require external specialized specification languages, and the theorem-prover translations are monadic, and only partially human-readable [12], [13], which increases verification costs. In contrast, our work is specific to ASL and PVS7, is not monadic, fully human-readable, and does not require other intermediate languages. [10] verifies that the binary and the source of an already formally verified OS kernel, seL4, have equivalent behaviors; the work uses SMT solvers for translation verification. This work is also restricted to a subset of C, $C_0$.

## VIII. Conclusions

We presented a methodology and a toolchain for translating functions from large binary codes, targeting the ARMv8 architecture, and formalizing them inside a theorem-prover.

Our translation verification methodology involves generating large-scale instruction emulation test lemmas, whose proof obligations are automatically discharged. We demonstrated our methodology and toolchain on Google's Zircon OS and a subset of Linux: we extracted 370 functions from these OSes, translated them into PVS7, and verified the correctness of translation by automatically generating thousands of proof obligations and tests. Our work demonstrates the feasibility of formalizing binary codes of large OSes with sources written in multiple languages and no formal semantics, with relatively low cost and a small trust base. Currently, our work has several limitations, all of which can be addressed in future work. We only formalized a subset of ARMv8.v3-A64 instructions (used in our targets' selected functions). We are also restricted to terminal functions (essential to formalizing almost all other functions). Additionally, we scope out concurrency, nondeterminism, and dynamically linked libraries. In the near future we plan to add functions with loops and function calls. Finally, we do not model system mode instructions since their ASL code is not currently fully open-source.

## Acknowledgment

This work was supported in part by ONR under grant N00014-18-1-2665.

## References

[1] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal verification of an OS kernel," in *ACM Symposium on Operating Systems Principles*. ACM, 2009, pp. 207–220.

[2] G. Heiser and K. Elphinstone, "L4 microkernels: The lessons from 20 years of research and deployment," *ACM Transactions on Computer Systems (TOCS)*, vol. 34, no. 1, p. 1, 2016.

[3] X. Leroy, "Formal verification of a realistic compiler," *Commun. ACM*, vol. 52, no. 7, pp. 107–115, Jul. 2009. [Online]. Available: http://doi.acm.org/10.1145/1538788.1538814

[4] A. Ahmed, A. W. Appel, C. D. Richards, K. N. Swadi, G. Tan, and D. C. Wang, "Semantic foundations for typed assembly languages," *ACM Trans. Program. Lang. Syst.*, vol. 32, no. 3, pp. 7:1–7:67, Mar. 2010. [Online]. Available: http://doi.acm.org/10.1145/1709093.1709094

[5] T. Sewell, F. Kam, and G. Heiser, "Complete, high-assurance determination of loop bounds and infeasible paths for wcet analysis," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2016 IEEE*. IEEE, 2016, pp. 1–11.

[6] N. Shankar, "Combining theorem proving and model checking through symbolic analysis," in *CONCUR 2000 - Concurrency Theory, 11th International Conference, University Park, PA, USA, August 22-25, 2000, Proceedings*, 2000, pp. 1–16. [Online]. Available: https://doi.org/10.1007/3-540-44618-4_1

[7] S. Owre, J. Rushby, N. Shankar, and D. Stringer-Calvert, "PVS: an experience report," in *Applied Formal Methods—FM-Trends 98*, ser. Lecture Notes in Computer Science, D. Hutter, W. Stephan, P. Traverso, and M. Ullman, Eds., vol. 1641. Boppard, Germany: Springer-Verlag, oct 1998, pp. 338–345. [Online]. Available: http://www.csl.sri.com/papers/fmtrends98/

[8] J. Ševčík, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell, "Compcerttso: A verified compiler for relaxed-memory concurrency," *J. ACM*, vol. 60, no. 3, pp. 22:1–22:50, Jun. 2013. [Online]. Available: http://doi.acm.org/10.1145/2487241.2487248

[9] L. Nelson, H. Sigurbjarnarson, K. Zhang, D. Johnson, J. Bornholt, E. Torlak, and X. Wang, "Hyperkernel: Push-button verification of an os kernel," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: ACM, 2017, pp. 252–269. [Online]. Available: http://doi.acm.org/10.1145/3132747.3132748

[10] T. A. L. Sewell, M. O. Myreen, and G. Klein, "Translation validation for a verified OS kernel," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, 2013, pp. 471–482. [Online]. Available: https://doi.org/10.1145/2491956.2462183

[11] S. Flur, K. E. Gray, C. Pulte, S. Sarkar, A. Sezgin, L. Maranget, W. Deacon, and P. Sewell, "Modelling the armv8 architecture, operationally: Concurrency and isa," *SIGPLAN Not.*, vol. 51, no. 1, pp. 608–621, Jan. 2016. [Online]. Available: http://doi.acm.org/10.1145/2914770.2837615

[12] A. C. J. Fox and M. O. Myreen, "A trustworthy monadic formalization of the armv7 instruction set architecture," in *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, 2010, pp. 243–258. [Online]. Available: https://doi.org/10.1007/978-3-642-14052-5_18

[13] A. C. J. Fox, "Improved tool support for machine-code decompilation in HOL4," in *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*, 2015, pp. 187–202. [Online]. Available: https://doi.org/10.1007/978-3-319-22102-1_12

[14] K. E. Gray, G. Kerneis, D. Mulligan, C. Pulte, S. Sarkar, and P. Sewell, "An integrated concurrency and core-isa architectural envelope definition, and test oracle, for ibm power multiprocessors," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. New York, NY, USA: ACM, 2015, pp. 635–646. [Online]. Available: http://doi.acm.org/10.1145/2830772.2830775

[15] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell, *Litmus: Running Tests against Hardware*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 41–44. [Online]. Available: https://doi.org/10.1007/978-3-642-19835-9_5

[16] S. Alvarez, *radare2*, radare, 2018, https://rada.re/r/.

[17] N. Shankar, S. Owre, and J. M. Rushby, *The PVS Proof Checker: A Reference Manual*, Computer Science Laboratory, SRI International, Menlo Park, CA, Feb. 1993, a new edition for PVS Version 2 is released in 1998.

[18] A. Reid, R. Chen, A. Deligiannis, D. Gilday, D. Hoyes, W. Keen, A. Pathirane, O. Shepherd, P. Vrabel, and A. Zaidi, "End-to-end verification of processors with isa-formal," in *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, 2016, pp. 42–58. [Online]. Available: https://doi.org/10.1007/978-3-319-41540-6_3

[19] A. Reid, "Trustworthy specifications of arm® v8-a and v8-m system level architecture," in *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*, 2016, pp. 161–168. [Online]. Available: https://doi.org/10.1109/FMCAD.2016.7886675

[20] N. A. Quynh and D. H. Vu, *Unicorn-The ultimate CPU emulator*, 2018, http://www.unicorn-engine.org/.

[21] A. Reid, "Trustworthy specifications of arm® v8-a and v8-m system level architecture," in *Proceedings of the 16th Conference on Formal Methods in Computer-Aided Design*, ser. FMCAD '16. Austin, TX: FMCAD Inc, 2016, pp. 161–168. [Online]. Available: http://dl.acm.org/citation.cfm?id=3077629.3077658

[22] *ARM v8 Manual*, ARM, 2019, https://static.docs.arm.com/ddi0553/a/DDI0553A_e_armv8m_arm.pdf.

[23] A. Armstrong, T. Bauereiss, B. Campbell, A. Reid, K. E. Gray, R. M. Norton, P. Mundkur, M. Wassell, J. French, C. Pulte, S. Flur, I. Stark, N. Krishnaswami, and P. Sewell, "Isa semantics for armv8-a, risc-v, and cheri-mips," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 71:1–71:31, Jan. 2019. [Online]. Available: http://doi.acm.org/10.1145/3290384

[24] C. Munoz, "Batch proving and proof scripting in PVS," *NIA-NASA Langley, National Institute of Aerospace, Hampton, VA, Report NIA Report*, no. 2007-03, 2007.

[25] A. P. Smith, C. A. Muñoz, A. J. Narkawicz, and M. Markevicius, "Kodiak: An implementation framework for branch and bound algorithms," 2015.

[26] X. Leroy, "Formal verification of a realistic compiler," *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, 2009. [Online]. Available: http://xavierleroy.org/publi/compcert-CACM.pdf

[27] A. Pnueli, M. Siegel, and E. Singerman, "Translation validation," in *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, ser. TACAS '98. Berlin, Heidelberg: Springer-Verlag, 1998, pp. 151–166. [Online]. Available: http://dl.acm.org/citation.cfm?id=646482.691453

[28] O. Grumberg, Ed., *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*, ser. Lecture Notes in Computer Science, vol. 1254. Springer, 1997. [Online]. Available: https://doi.org/10.1007/3-540-63166-6

[29] X. Leroy, "Formal verification of a realistic compiler," *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, 2009.

[30] G. C. Necula, "Translation validation for an optimizing compiler," in *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, ser. PLDI '00. New York, NY, USA: ACM, 2000, pp. 83–94. [Online]. Available: http://doi.acm.org/10.1145/349299.349314

[31] A. Pnueli, M. Siegel, and E. Singerman, "Translation validation," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 1998, pp. 151–166.

[32] W. Goerigk, "Towards Rigorous Compiler Implementation Verification," in *Proc. of the 1997 Workshop on Programming Languages and Fundamentals of Programming*, R. Berghammer and F. Simon, Eds., Avendorf, Germany, Nov. 1997, pp. 118 – 126.

[33] M. O. Myreen, A. C. J. Fox, and M. J. C. Gordon, "Hoare logic for ARM machine code," in *International Symposium on Fundamentals of Software Engineering, International Symposium, FSEN 2007, Tehran, Iran, April 17-19, 2007, Proceedings*, 2007, pp. 272–286. [Online]. Available: https://doi.org/10.1007/978-3-540-75698-9_18

[34] D. P. Mulligan, S. Owens, K. E. Gray, T. Ridge, and P. Sewell, "Lem: Reusable engineering of real-world semantics," *SIGPLAN Not.*, vol. 49, no. 9, pp. 175–188, Aug. 2014. [Online]. Available: http://doi.acm.org/10.1145/2692915.2628143